Strutture dati elementari

Pietro Di Lena

Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2021/2022

Introduzione

- Struttura dati:
  - Definisce come i dati sono logicamente organizzati
  - Definisce le operazioni per accedere e modificare i dati
- Descrive come i dati sono organizzati non quali dati sono memorizzati
  - Esempio: la struttura lista può contenere interi oppure stringhe
- Vedremo tre tipologie di strutture dati elementari
  - Liste concatenate (Linked List)
  - Pile (Stack)
  - Code (Queue)
  - Alberi (Tree)

Preliminari: prototipo vs implementazione

- Prototipo
  - Descrive possibili valori ed operazioni di una struttura dati
  - Nasconde i dettagli implementativi
  - Permette al programmatore di implementare la struttura dati
  - Permette all'utente di capire come usare la struttura dati
- Implementazione
  - Realizzazione di una struttura dati con qualche linguaggio di programmazione
  - Non visibile all'utente
  - Può avere un forte impatto sui tempi di esecuzione

Preliminari: classi di strutture dati

- Alcune classi di strutture dati
  - Lineari: dati in ordine sequenziale (primo elemento, secondo, ...)
  - Non-lineari: nessun ordine sequenziale

  - Statiche: numero di elementi costante
  - Dinamiche: il numero di elementi può variare dinamicamente

  - Omogenee: un solo tipo di dato memorizzabile (numeri, stringhe, ..)
  - Eterogenee: differenti tipi di dato memorizzabili

Esempio: struttura dati Dizionario

- Struttura dati generica per memorizzare oggetti
  - Contiene un insieme di chiavi univoche
  - Ogni chiave è associata ad un valore
  - I valori posso essere duplicati, le chiavi sono uniche

- Conosciuta anche come Array associativo

- Un Dizionario è un insieme dinamico
  - Il suo contenuto può crescere, contrarsi ed essere modificato

- Operazioni basilari di un Dizionario (prototipo):
  - SEARCH(Key k): cerca l'oggetto associato alla chiave k
  - INSERT(Key k, Data d): aggiunge la coppia (k, d) al Dizionario
  - DELETE(Key k): elmina la coppia (k, d) dal Dizionario

Java (asdlab.libreria.StruttureElem.Dizionario)

Interfaccia per la struttura dati Dizionario

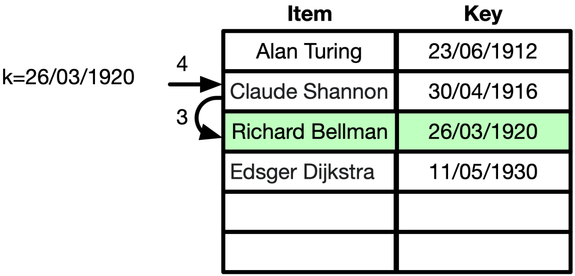
```
public interface Dizionario {

        // Aggiunge al dizionario la coppia (e,k)
        public void insert(Object e, Comparable k);


        // Rimuove dal dizionario l'elemento con chiave k
        public void delete(Comparable k);


        // Restituisce l'elemento e con chiave k
        public Object search(Comparable k);

}
```

Dizionario su array ordinato

- Idea: usiamo un array per salvare le coppie (Key,Data) e lo manteniamo ordinato rispetto a Key dopo inserimento e rimozione
  - Key può essere un intero (ordine numerico), una stringa (ordine lessicografico), una data (ordine numerico su tre valori), ...
- SEARCH(KEY k)
  - Cerca la chiave k con ricerca binaria sull'array ordinato
- INSERT(KEY k, DATA d)
  - Cerca con ricerca binaria (modificata) la posizione di k
  - Sposta di una posizione in avanti tutte le coppie con chiave $> k$
  - Inserisce la coppia (k, d) nello *spazio* aperto dallo spostamento
- DELETE(KEY k)
  - Cerca la chiave k con ricerca binaria
  - Elimina la coppia (k, d) dall'array
  - Sposta di una posizione indietro tutte le coppie con chiave $> k$

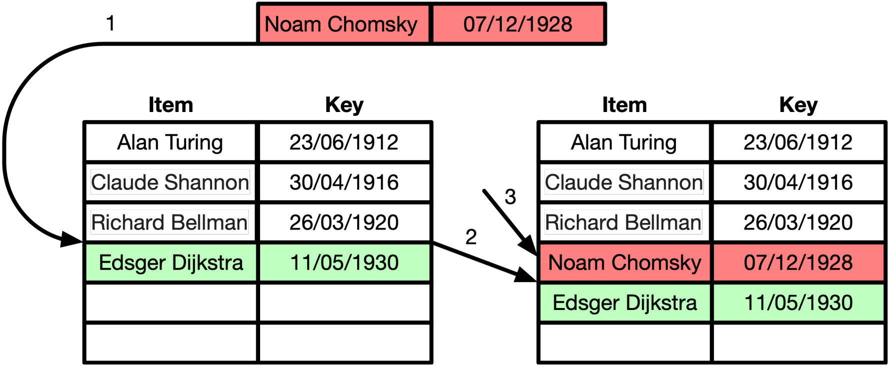
Implementazione di search su array ordinato

- Ricerca binaria della chiave k su array ordinato: $O(\log n)$
  1. Ispezioniamo la chiave k' nel centro dell'array
  2. Se $k' = NIL$ (array vuoto) allora l'oggetto non è nell'array
  3. Se $k = k'$ allora abbiamo trovato l'oggetto
  4. Se $k < k'$, ripetiamo la ricerca da 1 sulla prima metà dell'array
  5. Se $k > k'$, ripetiamo la ricerca da 1 sulla seconda metà dell'array



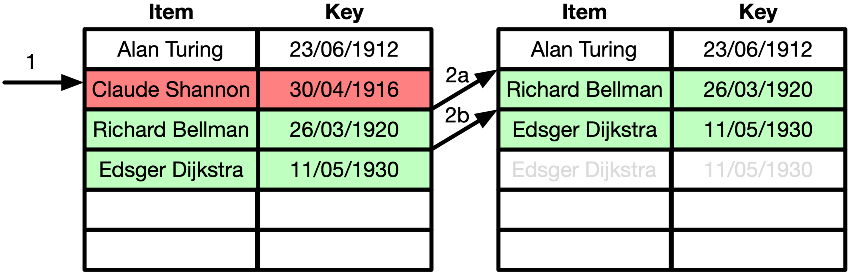
Implementazione di insert su array ordinato

- Inserimento su array ordinato:
  1. Ricerca della posizione in cui inserire la chiave k: $O(\log n)$
  2. Spostamento in avanti di tutte le coppie con chiave $> k$: $O(n)$
  3. Inserimento della coppia (k, d) nello spazio creato: $O(1)$



Implementazione di delete su array ordinato

- Rimozione su array ordinato:
  1. Ricerca binaria della chiave k: $O(\log n)$
  2. Spostamento indietro di tutte le coppie con chiave $> k$: $O(n)$



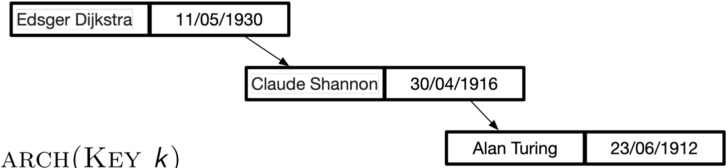
- N.B. Sovrascriviamo (non eliminiamo) la coppia con chiave k
- N.B. Non abbiamo bisogno di cancellare il contenuto dell'ultima posizione precedentemente occupata nell'array

Costo delle operazioni su array ordinato

- SEARCH(Key k)
  - Ricerca binaria su array ordinato
  - Costo nel caso pessimo: $O(\log n)$
- INSERT(Key k, Data d)
  - Ricerca binaria modificata + spostamento + inserimento
  - Costo nel caso pessimo: $O(\log n) + O(n) + O(1) = O(n)$
- DELETE(Key k)
  - Ricerca binaria della chiave + spostamento
  - Costo nel caso pessimo: $O(\log n) + O(n) = O(n)$

Dizionario su lista concatenata

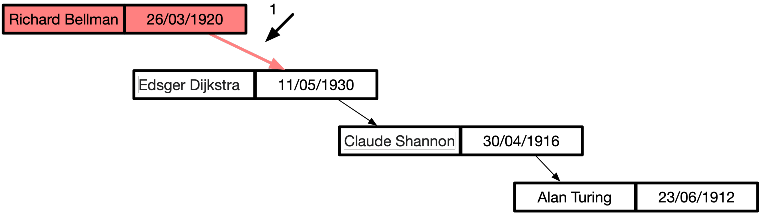
- Idea: lista concatenata non ordinata per memorizzare le coppie
  - Ogni coppia punta alla successiva nella lista
  - Per cercare una chiave bisogna visitare la lista dal primo elemento
  - Ogni nuova coppia può essere inserita in testa alla lista



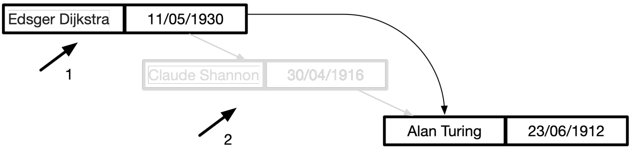
- SEARCH(KEY k)
  - Cerca la chiave k con ricerca sequenziale partendo dalla testa
- INSERT(KEY k, DATA d)
  - Inserisce la nuova coppia in testa alla lista
- DELETE(KEY k)
  - Cerca la chiave k con ricerca sequenziale
  - Elimina il nodo relativo alla coppia (k, d) dalla lista

Implementazione di insert/delete su lista

- Inserimento su lista concatenata:
  1. Inseriamo la nuova coppia in testa alla lista: $O(1)$



- Rimozione su lista concatenata:
  1. Cerchiamo la chiave con ricerca sequenziale: $O(n)$
  2. Rimuoviamo la coppia associata con la chiave: $O(1)$



Cost delle operazioni su lista concatenata

- SEARCH(Key k)
  - Ricerca sequenziale su lista non ordinata
  - Costo nel caso pessimo: $O(n)$
- INSERT(Key k, Data d)
  - Inserimento in testa alla lista
  - Costo nel caso pessimo: $O(1)$
- DELETE(Key k)
  - Ricerca sequenziale + rimozione
  - Costo nel caso pessimo: $O(n) + O(1) = O(n)$
- Domanda: perchè non manteniamo la lista ordinata?

Conclusioni: Array vs Liste concatenate

- Siamo partiti dal prototipo della struttura dati Dizionario

- Due strategie implementative differenti portano a prestazioni differenti

Funzione	Array ordinato	Lista concatenata
SEARCH	$O(\log n)$	$O(n)$
INSERT	$O(n)$	$O(1)$
DELETE	$O(n)$	$O(n)$

- Inoltre
  - Le Liste concatenate hanno una dimensione dinamica
  - Gli Array hanno una dimensione fissa

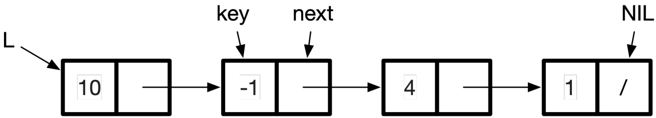
- Non possiamo dire quale delle due implementazioni sia la migliore
  - La scelta migliore dipende dall'applicazione che richiede un Dizionario come struttura dati di supporto

Strutture dati elementari: Liste concatenate

- Una Lista è una struttura dati in cui tutti gli elementi sono organizzati in ordine sequenziale
- Una lista supporta almeno tre operazioni basilari:
  - Ricerca, Inserimento, Rimozione
- Implementazione con Array
  - L'ordine sequenziale è determinato dagli indici dell'array
  - Lo spazio per gli elementi è allocato staticamente
  - Spazio limitato ma accesso veloce agli elementi
- Implementazione con Liste concatenate (Linked Lists)
  - L'ordinamento è determinato da una catena di puntatori
  - Lo spazio per gli elementi è allocato dinamicamente su richiesta
  - Accesso lento ma dimensione illimitata
- Ci concentriamo su diversi tipi di Liste concatenate

Lista concatenata semplice

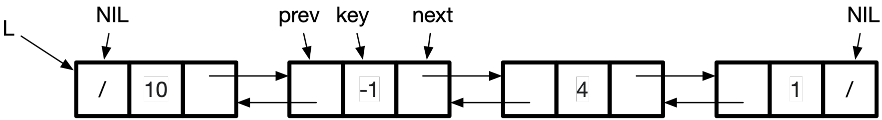
- Ogni nodo x di una Lista concatenata semplice contiene
  - $x.key$: un valore chiave (non necessariamente unico)
  - $x.next$: un puntatore al nodo successivo nella lista
- Se $x.next = NIL$ allora x è l'ultimo nodo nella lista
- Un nodo può contenere altri dati oltre alla chiave (Es. $x.data$)



- Può essere visitata in un'unica direzione (dalla testa verso la coda)
- In inglese nota come Singly Linked List

Lista doppiamente concatenata

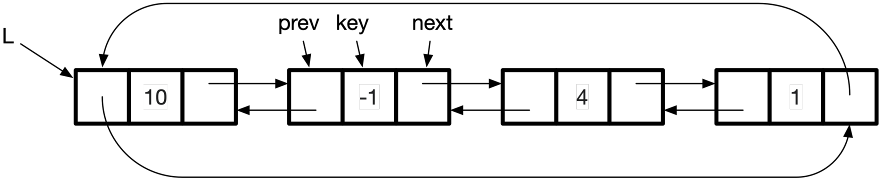
- Sono Liste concatenate semplici in cui ogni nodo contiene anche
  - $x.prev$: un puntatore al nodo precedente nella lista
- Se $x.prev = NIL$ allora x è il primo nodo nella lista



- Può essere visitata in entrambe le direzioni
- In inglese nota come Doubly Linked List

Lista concatenata circolare

- Solo Liste doppiamente concatenate in cui
  - Il campo *next* dell'ultimo nodo punta al primo nodo
  - Il campo *prev* del primo nodo punta all'ultimo nodo



- Può essere visitata in entrambe le direzioni
- L'accesso alla testa dalla coda è veloce (cosi come il contrario)
- In inglese nota come Circular Linked List

SEARCH SUL LISTA CONCATENATA CIRCOLARE

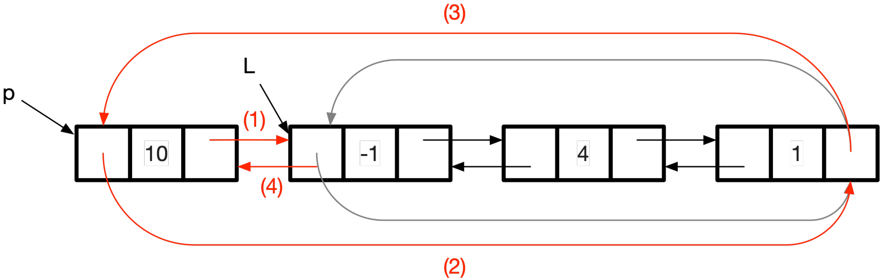
```
1: function SEARCH(LIST L, INT x) → LIST
2:     if L ≠ NIL then
3:         LIST p = L
4:         repeat
5:             if p.key == x then
6:                 return p
7:             p = p.next
8:         until p ≠ L
9:     return NIL
```

- Costo nel caso pessimo: O(n) (loop repeat nelle linee 4-8)
    - $n =$ numero di nodi nella lista
- Costo nel caso ottimo: O(1) (il nodo da cercare è in testa)

INSERT su Lista concatenata circolare

```
function INSERT(List L, Int x) → List
    List p = NEW List(x)
    p.next = p.prev = p            ▷ points to itself
    if L ≠ NIL then
        p.next = L                 ▷ Update (1)
        p.prev = L.prev            ▷ Update (2)
        L.prev.next = p            ▷ Update (3)
        L.prev = p                 ▷ Update (4)
    return p
```

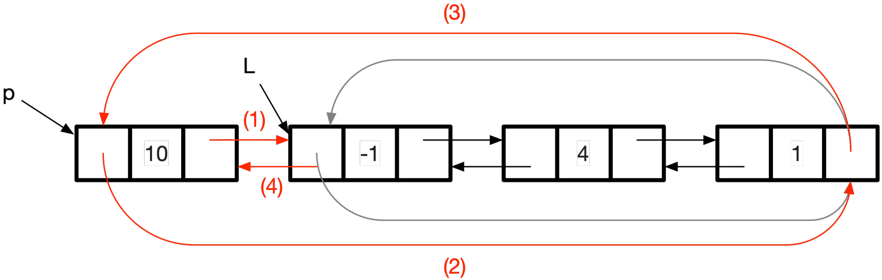
- Inserimento in testa
- Ritorna nuova testa
- Caso pessimo:
- Caso ottimo:



INSERT SU LISTA CONCATENATA CIRCOLARE

```
function INSERT(LIST L, INT x) → LIST
    LIST p = NEW LIST(x)
    p.next = p.prev = p              ▷ points to itself
    if L ≠ NIL then
        p.next = L                   ▷ Update (1)
        p.prev = L.prev              ▷ Update (2)
        L.prev.next = p              ▷ Update (3)
        L.prev = p                   ▷ Update (4)
    return p
```

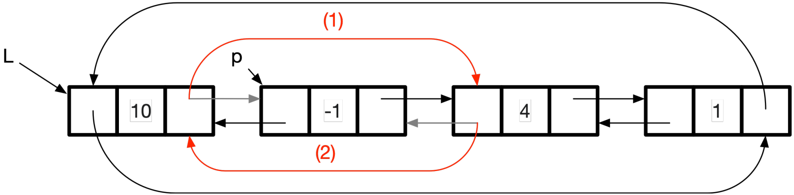
- Inserimento in testa
- Ritorna nuova testa
- Caso pessimo: $O(1)$
- Caso ottimo: $O(1)$



delete su Lista concatenata circolare

```
1:  function DELETE(LIST L, INT x) → LIST
2:      LIST p = SEARCH(L, x)
3:      if p ≠ NIL then
4:          p.prev.next = p.next        ▷ Update (1)
5:          p.next.prev = p.prev        ▷ Update (2)
6:          if p == L then              ▷ Head delete
7:              if L.next == L then
8:                  L = NIL             ▷ Single element
9:              else
10:                 L = p.next
11:         FREE(p)
12:     return L
```

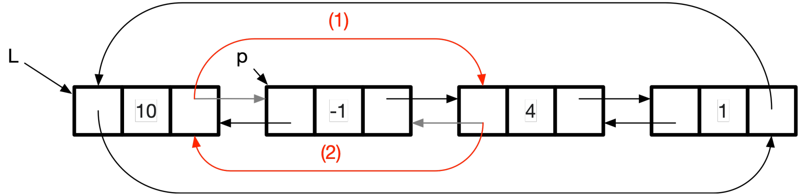
- Ritorna nuova testa
- SEARCH (2): $O(n)$
- Update (3-10): $O(1)$
- FREE (11): $O(1)$
- Caso pessimo:
- Caso ottimo:



DELETE su Lista concatenata circolare

```
1:  function DELETE(LIST L, INT x) → LIST
2:      LIST p = SEARCH(L, x)
3:      if p ≠ NIL then
4:          p.prev.next = p.next          ▷ Update (1)
5:          p.next.prev = p.prev          ▷ Update (2)
6:          if p == L then                ▷ Head delete
7:              if L.next == L then
8:                  L = NIL               ▷ Single element
9:              else
10:                 L = p.next
11:         FREE(p)
12:     return L
```

- Ritorna nuova testa
- SEARCH (2): $O(n)$
- Update (3-10): $O(1)$
- FREE (11): $O(1)$
- Caso pessimo: $O(n)$
- Caso ottimo: $O(1)$



Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Classe per una lista concatenata circolare

```
// Implementazione basata su lista circolare
public class StrutturaCollegata implements Dizionario {
        private Record list = null;

        private final class Record { ... }

        public void insert(Object e, Comparable k)
        { ... }

        public void delete(Comparable k)
        { ... }

        public Object search(Comparable k)
        { ... }

}
```

Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Classe per definire un nodo della lista

```
private final class Record {
        public Object     elem;
        public Comparable chiave;
        public Record     next;
        public Record     prev;

         public Record(Object e, Comparable k) {
                elem = e;
                chiave = k;
                next = prev = null;
        }
}
```

Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Metodo per l'inserimento

```
 public void insert(Object e, Comparable k) {
        Record p = new Record(e, k);
        if (list == null)
                list = p.prev = p.next = p;
        else {
                p.next = list.next;
                list.next.prev = p;
                list.next = p;
                p.prev = list;
        }
}
```

N.B. list punta all'ultimo nodo nella Lista concatenata

Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Metodo per la ricerca

```
public Object search(Comparable k) {
        if (list == null) return null;
        for (Record p = list.next; ; p = p.next){
                if (p.chiave.equals(k)) return p.elem;
                if (p == list) return null;
        }
}
```

N.B. La ricerca parte dal secondo nodo nella lista

Java (asdlab.libreria.StruttureElem.StrutturaCollegata)

Metodo per la rimozione

```
public void delete(Comparable k) {
        Record p = null;
        if (list != null)
                for (p = list.next; ; p = p.next) {
                        if (p.chiave.equals(k)) break;
                        if (p == list) { p = null; break;}
                }
        if (p == null)
                throw new EccezioneChiaveNonValida();
        if (p.next == p) list = null;
        else {
                if (list == p) list = p.next;
                p.next.prev = p.prev;
                p.prev.next = p.next;
        }
}
```

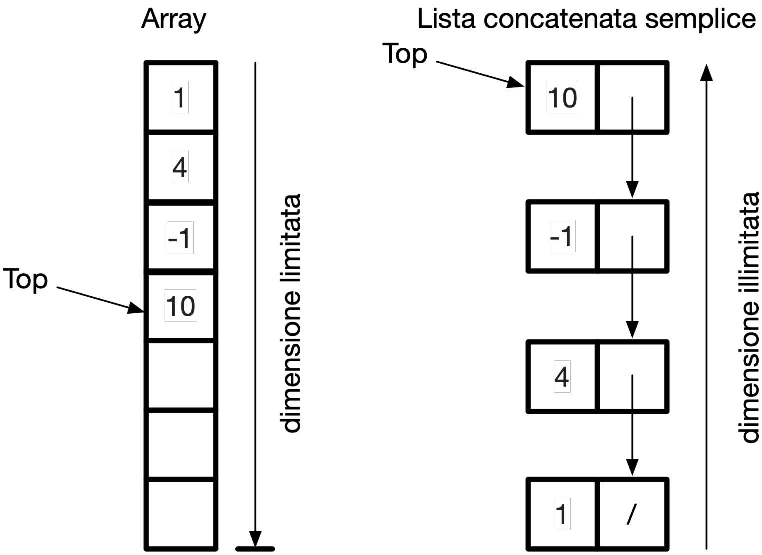
Strutture dati elementari: Pila (Stack)

- Una Pila è una struttura dati che supporta due operazioni principali
  - PUSH: aggiunge un nuovo elemento alla struttura
  - POP: rimuove l'elemento aggiunto più di recente
- Intuitivamente, una pila di elementi uno un cima all'altro
  - Ad esempio, una pila di piatti
  - Modalità LIFO (Last In First Out)
- Applicazioni delle pile
  - Gestione di record di attivazione (chiamate a funzione sul calcolatore)
  - Linguaggi stack-oriented (PostScript, BibTex, ...)
  - Numerose applicazioni in algoritmi
    - Editor di testo (operazioni undo e redo)
    - Syntax parsing (parentesi bilanciate)
    - ...

Implementazione della struttura dati Pila

- Una Pila è una Lista che supporta un numero limitato di operazioni
- Implementazione con Liste concatenate semplici (esercizio)
  - POP: rimuove la testa della lista
  - PUSH: inserisce l'elemento in testa alla lista
  - Pro: dimensione illimitata
  - Con: piccolo overhead di memoria (valore+puntatore)
  - Domanda: perchè non usare Liste doppiamente concatenate?
- Implementazione con Array
  - POP: rimuove l'ultimo elemento nell'array
  - PUSH: inserisce l'elemento nella prima posizione libera
  - Pro: nessun overhead di memoria (memorizza solo il valore)
  - Con: dimensione limitata
- In entrambi i casi POP and PUSH costano $O(1)$

Implementazione della struttura dati Pila

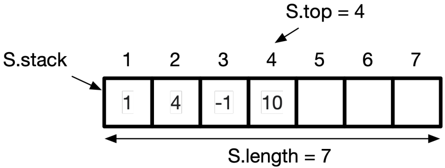


PUSH AND POP CON ARRAY STATICO

```
function PUSH(STACK S, INT x)
    if S.top == S.length then
        error "overflow"
    else
        S.top = S.top + 1
        S.stack[S.top] = x
```

```
function POP(STACK S) → INT
    if S.top == 0 then
        error "underflow"
    else
        e = S.stack[S.top]
        S.top = S.top − 1
        return e
```

- Entrambe le funzioni costano ?
- Stack underflow causato da un uso poco attento di POP
- Stack overflow causato da mancanza di spazio
- Come implementare una Pila con array dinamico?

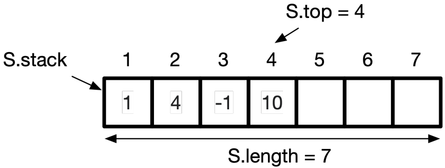


PUSH AND POP CON ARRAY STATICO

```
function PUSH(STACK S, INT x)
    if S.top == S.length then
        error "overflow"
    else
        S.top = S.top + 1
        S.stack[S.top] = x
```

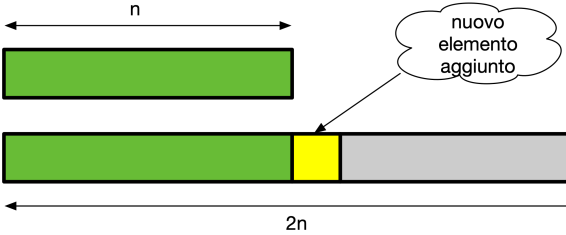
```
function POP(STACK S) → INT
    if S.top == 0 then
        error "underflow"
    else
        e = S.stack[S.top]
        S.top = S.top − 1
        return e
```

- Entrambe le funzioni costano (pessimo e ottimo) $O(1)$
- Stack underflow causato da un uso poco attento di POP
- Stack overflow causato da mancanza di spazio
- Come implementare una Pila con array dinamico?

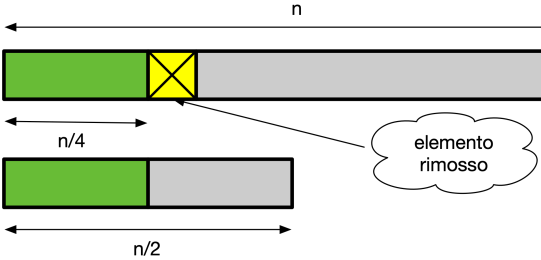


Strategia con array dinamico

- Raddoppiamo la dimensione dell'array quando non c'è spazio libero



- Dimezziamo la dimensione dell'array quando l'occupazione è di 1/4



POP CON ARRAY DINAMICO

```
1:  function POP(STACK S) → INT
2:      if S.top == 0 then
3:          error "underflow"
4:      else
5:          e = S.stack[S.top]
6:          S.top = S.top − 1
7:          if S.top ≤ ⌊S.length/4⌋ then
8:              n = S.length
9:              LET T[1 ··· ⌈n/2⌉] BE A NEW ARRAY
10:             for i = 1, ··· , ⌊n/4⌋ do
11:                 T[i] = S.stack[i]
12:             S.stack = T
13:             S.length = ⌈n/2⌉
14:         return e
```

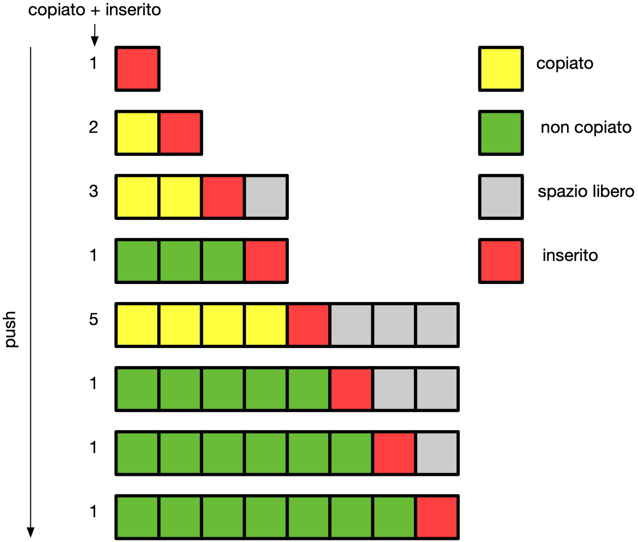
- Costo nel caso pessimo: $O(n)$ (copia dell'array, linee $10 - 11$)
- Costo nel caso ottimo: $O(1)$ (array utilizzato per più di $1/4$)

PUSH CON ARRAY DINAMICO

```
1:  function PUSH(STACK S, INT x)
2:      if S.top == S.length then
3:          n = S.length
4:          LET T[1 ··· 2n] BE A NEW ARRAY
5:          for i = 1, ··· , n do
6:              T[i] = S.stack[i]
7:          S.stack = T
8:          S.length = 2n
9:      S.top = S.top + 1
10:     S.stack[S.top] = x
```

- Costo nel caso pessimo: $O(n)$ (copia dell'array, linee $5-6$)
- Costo nel caso ottimo: $O(1)$ (array non pieno)

Analisi di push



Analisi ammortizzata di push

- Costo nel caso pessimo $O(n)$, costo nel caso ottimo $O(1)$

- Qual è il costo di n PUSH partendo da una Pila vuota?
  - Il costo nel caso pessimo è $O(n^2)$ se usiamo l'upper bound $O(n)$
  - $O(n^2)$ non è una stima accurata: non raddoppiamo spesso l'array
  - Il costo dell'i-esima PUSH è infatti

$$c_i = \begin{cases} i & \text{se } i-1 \text{ è una potenza esatta di } 2 \\ 1 & \text{altrimenti} \end{cases}$$

  - Metodo degli aggregati: il costo totale di n PUSH è

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log_2 n} 2^j = n + \frac{2^{\log_2 n+1} - 1}{2-1} = 3n - 1 = O(n)$$

- Il costo ammortizzato di n PUSH è dunque $\dfrac{O(n)}{n} = O(1)$

Analisi ammortizzata di PUSH e POP

- Cosi come abbiamo fatto per PUSH possiamo dimostrare che il costo ammortizzato di n POP partendo da una Pila piena è $O(1)$
  - Non abbiamo una caratterizzazione semplice come per PUSH
  - Dobbiamo utilizzare il metodo degli accantonamenti
    - Costo ammortizzato: 2€ (1€ rimozione+1€ credito per copia)

- Quanto costa una generica sequenza di n PUSH e POP?
  - Di nuovo, metodo degli accantonamenti (non aggregati)
  - Possiamo dimostrare che il costo di ogni sequenza di n PUSH e POP su array dinamico inizialmente vuoto è al peggio $O(n)$ $\Rightarrow$ costo ammortizzato di $O(1)$ per entrambe PUSH e POP

- N.B. Tali costi ammortizzati valgono solo per la nostra strategia di espansione/contrazione ma non per ogni possibile strategia
  - Ad esempio, dimezzare l'array quando è pieno per metà porta a costi ammortizzati di $O(n)$ per operazione

Java (asdlab.libreria.StruttureElem.Pila)

Interfaccia Pila

```
public interface Pila {
        /**
         * Verifica se la pila è vuota.
         */
        public boolean isEmpty();
        /**
         * Aggiunge l'elemento in cima
         */
        public void push(Object e);
        /**
         * Restituisce l'elemento in cima
         */
        public Object top();
        /**
         * Cancella l'elemento in cima
         */
        public Object pop();
}
```

Java (asdlab.libreria.StruttureElem.PilaArray)

Implementazione con array dinamico

```
public class PilaArray implements Pila {
        private Object[] S = new Object[1];
        private int n = 0;

        public boolean isEmpty()
        {...}

        public void push(Object e)
        { ... }

        public Object top()
        { ... }

        public Object pop()
        { ... }
}
```

Java (asdlab.libreria.StruttureElem.PilaArray)

Metodo push

```
public void push(Object e) {
        if (n == S.length) {
                Object[] temp = new Object[2 * S.length];
                for (int i = 0; i < n; i++) temp[i] = S[i];
                S = temp;
        }
        S[n] = e;
        n = n + 1;
}
```

Java (asdlab.libreria.StruttureElem.PilaArray)

Metodo pop

```
public Object pop() {
    if (this.isEmpty())
        throw new EccezioneStrutturaVuota("Pila vuota");
    n = n - 1;
    Object e = S[n];
    if (n > 1 && n == S.length / 4) {
        Object[] temp = new Object[S.length / 2];
        for (int i = 0; i < n; i++) temp[i] = S[i];
        S = temp;
    }
    return e;
}

public boolean isEmpty() {
    return n == 0;
}
```

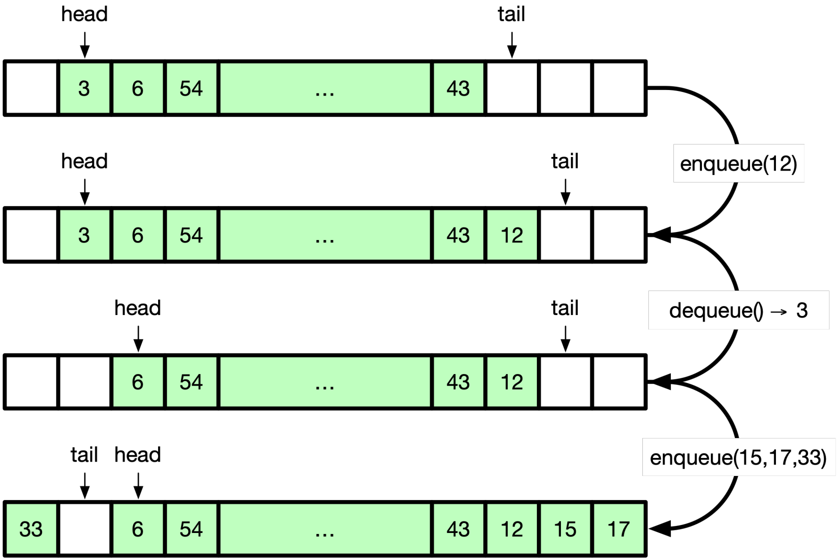
Strutture dati elementari: Coda (Queue)

- Una Coda è una struttura dati che supporta due operazioni principali
  - ENQUEUE: aggiunge un elemento in fondo alla coda
  - DEQUEUE: rimuove l'elemento in testa alla coda
- Gli elementi sono rimossi nello stesso ordine in cui sono inseriti
  - Non è possibile accedere ad elementi nel mezzo della Coda
- Intuitivamente, una coda rappresenta una fila di elementi
  - Ad esempio, una fila di persone in attesa di un servizio
  - Modalità FIFO (First In First Out)
- Applicazioni della code
  - Scheduling dei processi nei sistemi operativi
  - Visita di tipo Bread-first-search su grafi
  - ...

Implementazione della struttura dati Coda

- Una Coda è una Lista che supporta un numero limitato di operazioni
- Implementazione con Liste concatenate circolari (esercizio)
  - ENQUEUE: inserisce l'elemento in coda alla lista
  - DEQUEUE: rimuove la testa della lista
  - Pro: dimensione illimitata
  - Con: overhead di memoria (valore+2 puntatori)
- Implementazione con Liste concatenate semplici (esercizio)
  - Dobbiamo introdurre un puntatore alla coda
  - ENQUEUE: inserisce in coda usando il puntatore alla coda
  - Pro e Con come per Liste concatenate circolari
- Implementazione con Array circolari
  - Pro: nessun overhead di memoria (memorizza solo il valore)
  - Con: dimensione limitata
- In tutti e tre i casi ENQUEUE and DEQUEUE costano $O(1)$

Implementazione con array circolare



Implementazione con array circolare

```
function ENQUEUE(QUEUE Q, INT x)
    if Q.size == Q.length then
        error "overflow"
    Q, buf[Q.tail] = x
    Q.tail = (Q.tail%Q.length) + 1
    Q.size = Q.size + 1
```

```
function DEQUEUE(QUEUE Q) → INT
    if Q.size == 0 then                ▷ Empty Q
        error "underflow"
    x = Q.buf[Q.head]
    Q.head = (Q.head%Q.length) + 1
    Q.size = Q.size − 1
    return x
```

- Entrambe costano $O(1)$
- $Q.size$ è il numero di elementi in Q
- *tail* punta alla prima cella libera
- $\% =$ operazione modulo
- Il modulo permette una visita circolare dell'array
- Come implementare una struttura dati Coda con array dinamico circolare?

Java (asdlab.libreria.StruttureElem.Coda)

Interfaccia Coda

```
public interface Coda {
        //Verifica se la coda è vuota.
        public boolean isEmpty();

        //Aggiunge l'elemento in fondo alla coda
        public void enqueue(Object e);

        //Restituisce il primo elemento della coda
        public Object first();

        //Cancella il primo elemento nella coda
        public Object dequeue();
}
```

Implementazione con liste concatenate: CodaCollegata.java

Java: coda con array circolare 1/2

Implementazione con array circolare (non in asdlab)

```
public class CodaArrayCircolare implements Coda {
    private Object[] buffer; // Array di oggetti
    private int head;        // Dequeuing index
    private int tail;        // Enqueuing index
    private int size;        // Numero di elementi nella coda

    public CodaArrayCircolare(int max) {
      buffer = new Object[max];
      head = tail = size = 0;
    }

    @Override
    public boolean isEmpty() { return (size==0); }

    @Override
    public Object first() throws EccezioneStrutturaVuota {
      if (size == 0) throw new EccezioneStrutturaVuota("Coda vuota");
      else return buffer[head];
    }
    ...
```

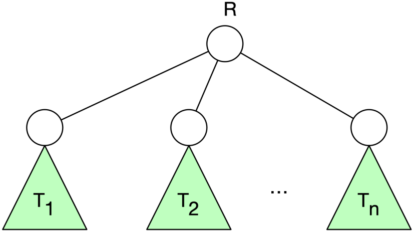
Java: coda con array circolare 2/2

Implementazione con array circolare (non in asdlab)

```
    @Override
    public void enqueue(Object o) {
      if (size == buffer.length)
           throw new EccezioneArrayPieno("Coda piena");
        buffer[tail] = o;
        tail =(tail+1) % buffer.length;
        size++;
    }

    @Override
    public Object dequeue() {
      if (size == 0) throw new EccezioneStrutturaVuota("Coda vuota");
        Object res = buffer[head];
        head =(head+1) % buffer.length;
        size--;
        return res;
    }
}
```

Strutture dati elementari: Alberi

- Un Albero è una struttura dati non-lineare ad albero gerarchico
- Definizione di struttura dati Albero:
  - Un insieme di nodi (or vertici)
  - Un insieme di archi che connettono nodi
  - Esiste un solo percorso per andare da un nodo all'altro
- Un Albero è ordinato se i figli di ogni nodo sono ordinati
  - Possiamo identificare il primo figlio, il secondo figlio, ...
- Un albero è radicato se uno dei suoi nodi è identificato come radice

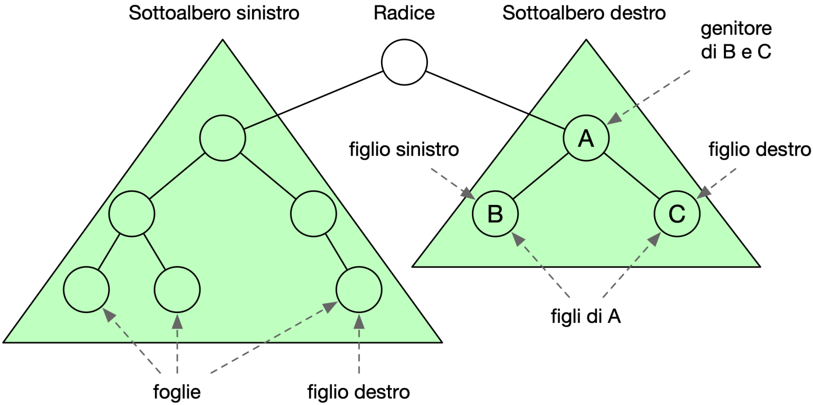


Definizione ricorsiva di Albero radicato:

- Insieme vuoto di nodi oppure
- Una radice R e zero o più alberi disgiunti (sotto-alberi) le cui radici sono connesse ad R

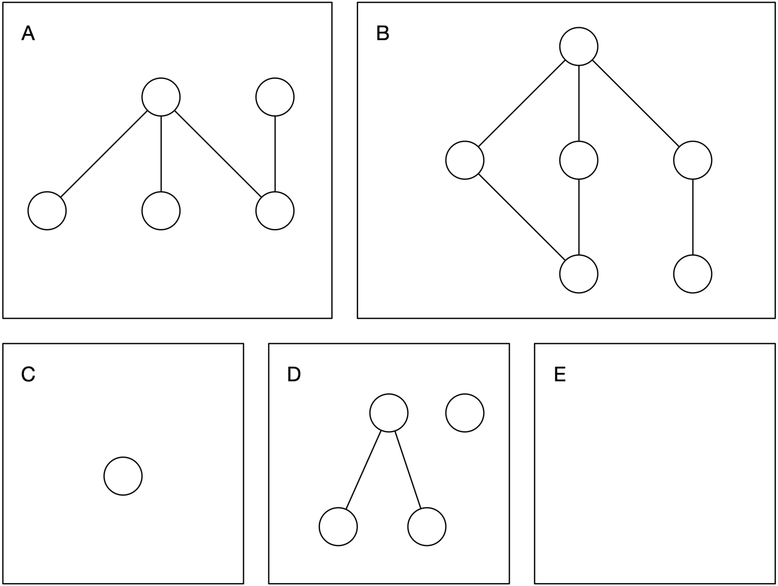
Alberi Binari

- Un Albero Binario è un Albero in cui ogni nodo ha al massimo due figli



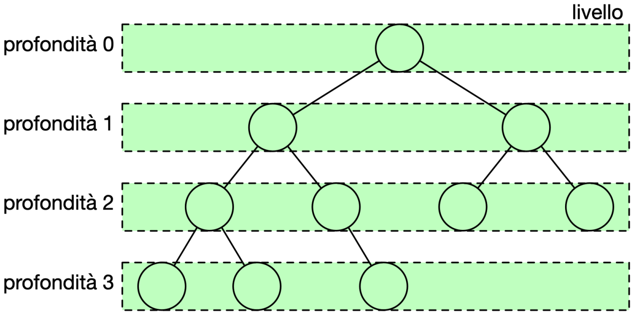
- Un Albero Binario è un Albero ordinato
  - Ogni nodo può avere un figlio sinistro e/o un figlio destro
  - Un nodo può avere figlio destro ma non sinistro (e al contrario)

Quali sono Alberi (Binari)?



Alcune definizioni

- La profondità di un nodo u è la lunghezza del percorso (unico) che va dalla radice al nodo u (numero di archi)
- Un livello è l'insieme di tutti i nodi alla stessa profondità
- L'altezza di un Albero è la sua massima profondità
- Il grado di un nodo è il numero dei suoi figli

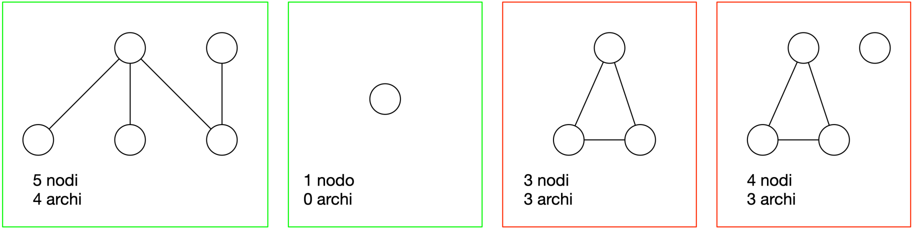


Proprietà fondamentale di un Albero

Teorema

Ogni Albero non-vuoto con n nodi ha esattamente $n-1$ archi
(Dimostrazione per induzione)

- Possiamo usare tale proprietà per dimostrare che una struttura dati non è un Albero
- Il Teorema non funziona nella direzione opposta

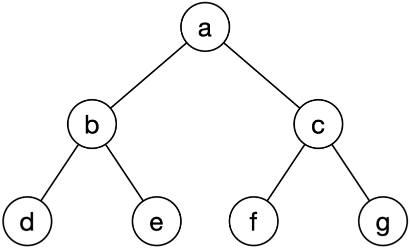


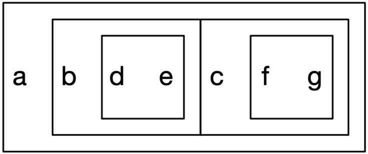
Algoritmi di visita su Alberi

- Algoritmo di visita (o anche *algoritmo di ricerca*) su Albero
  - Algoritmo per *visitare* tutti i nodi di una struttura dati Albero
- Visita in profondità o Depth-First Search (DFS)
  - La ricerca va in profondià il più possibile prima di visitare il nodo successivo nello stesso livello
  - Esistono tre varianti: pre-ordine, post-ordine, in-ordine
- Visita in ampiezza o Breadth-First Search (BFS)
  - La ricerca viene eseguita livello per livello

Visita in profondità: pre-ordine (pre-order)

```
function PREORDER(TREE T)
    if T ≠ NIL then
        VISIT(T)
        PREORDER(T.left)
        PREORDER(T.right)
```

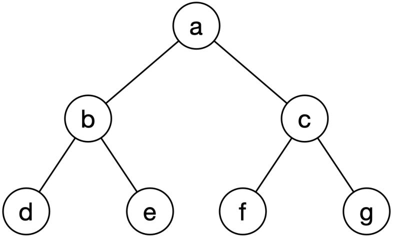


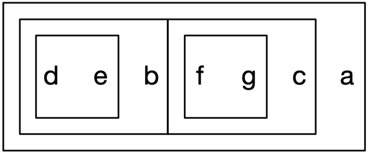


Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

Visita in profondità: post-ordine (post-order)

```
function POSTORDER(TREE T)
    if T ≠ NIL then
        POSTORDER(T.left)
        POSTORDER(T.right)
        VISIT(T)
```

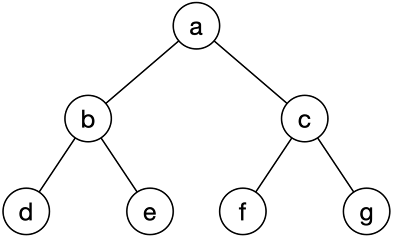


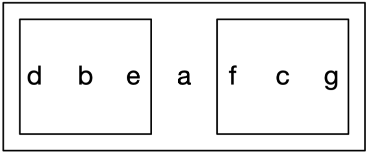


Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

VISITA IN PROFONDITÀ: IN-ORDINE (IN-ORDER)

```
function INORDER(TREE T)
    if T ≠ NIL then
        INORDER(T.left)
        VISIT(T)
        INORDER(T.right)
```

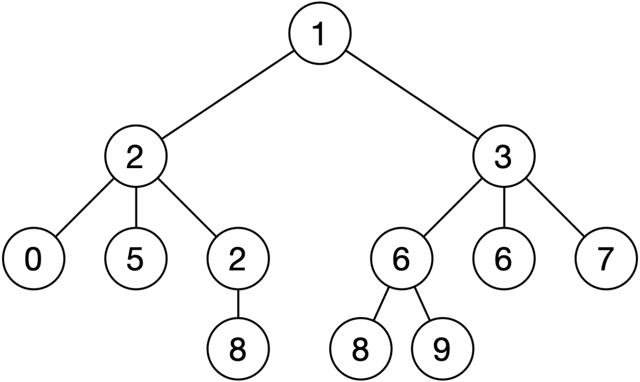




Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

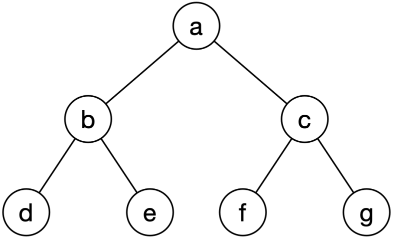
Visita in profonditàsu Alberi non-binari

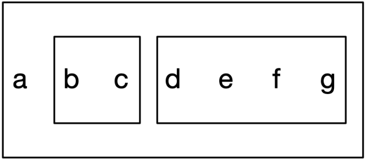
- Gli algoritmi di visita possono essere generalizzati ad Alberi non Binari
- Solo la visita in-ordine richiede specifiche aggiuntive
  - Es: visitiamo i primi *k* figli, poi nodo corrente, poi i figli rimanenti
- Qual è l'ordine di visita (pre,post,in) nel seguente Albero?



Visita in ampiezza (BFS)

```
function BFS(Tree T)
    Let Q be a new Queue
    if T ≠ NIL then
        ENQUEUE(Q, T)
    while Q.size ≠ 0 do
        x = DEQUEUE(Q)
        VISIT(x)
        if x.left ≠ NIL then
            ENQUEUE(Q, x.left)
        if x.right ≠ NIL then
            ENQUEUE(Q, x.right)
```

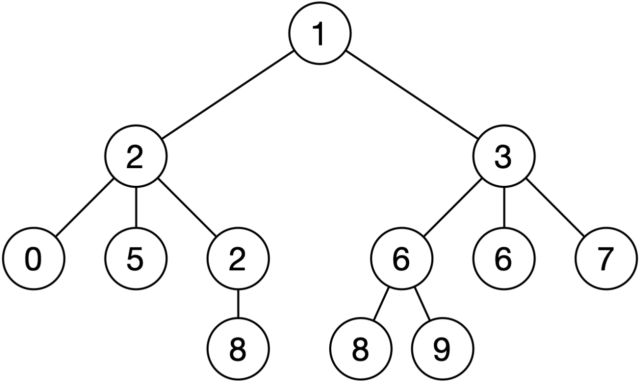




- N.B. Usiamo una Coda per imporre un ordine di visita per livello
- Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

Visita in ampiezza su Alberi non-binari

- Anche la visita in ampiezza è generalizzabile ad Alberi non-binari
- Qual è l'ordine di visita in ampiezza dei nodi nel seguente Albero?



Esempio: algoritmo su Albero Binario

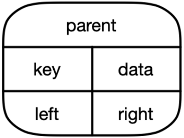
Scrivere un algoritmo per contare i nodi di un Albero Binario

```
function COUNTNODES(TREE T) → INT
    if T == NIL then
        return 0
    else
        return 1+COUNTNODES(T.left)+COUNTNODES(T.right)
```

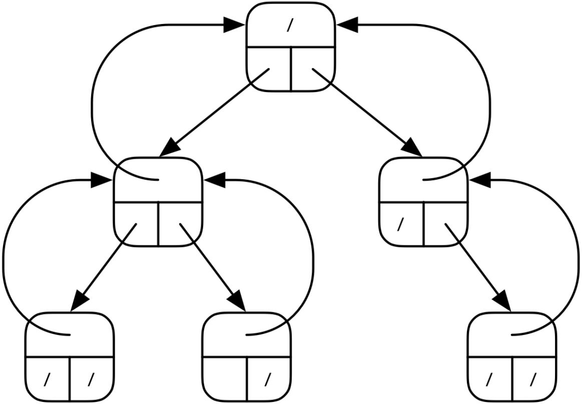
- Quale tipo di visita è implementato in COUNTNODES?
  - post-ordine (per valutare T bisogna prima valutare i suoi figli)
- Qual è il tempo di calcolo di COUNTNODES?
  - Costo (ottimo,pessimo,medio): $\Theta(n)$
- Come modificare l'Albero Binario in modo che COUNTNODES sia $O(1)$?
  - Aggiungiamo un campo *T.totnodes* ad ogni nodo
- Qual è l'impatto di tale modifica sul costo di altre operazioni, ad esempio *aggiungi una nuova foglia*, su un Albero Binario?

Implementazione di un Albero Binario

Nodo



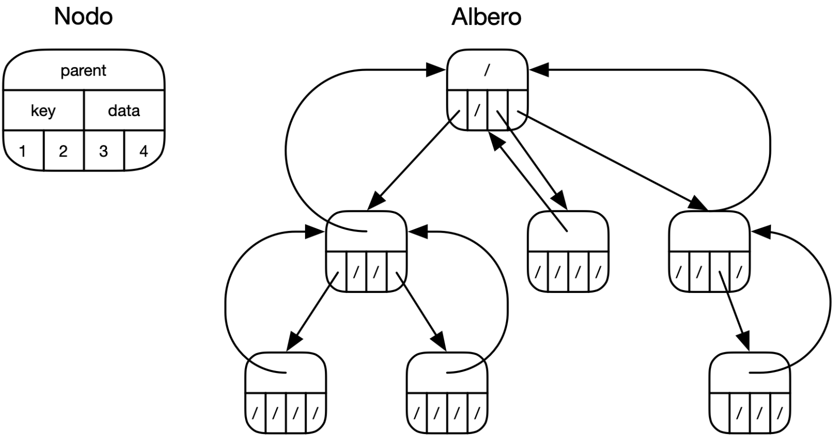
Albero Binario



Implementazione con puntatori

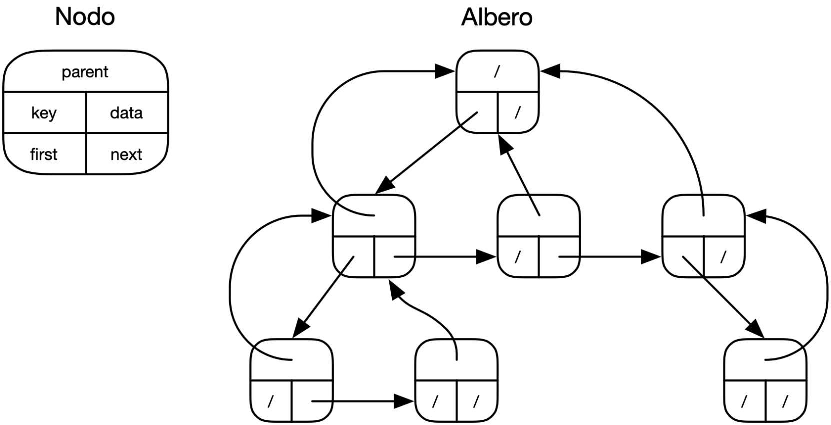
- left: puntatore al figlio sinistro
- right: puntatore al figlio destro

Implementazione di un Albero non-binario 1



- Ogni nodo contiene un array di puntatori a k figli
- Il numero massimo di k figli è fisso
- Rischio di sprecare spazio se molti nodi hanno meno di k figli

Implementazione di un Albero non-binario 2



- Ogni nodo ha un puntatore al primo (first) figlio
- Ogni nodo ha un puntatore al fratello successivo (next)
- La lista di figli è gestita con una Lista concatenata sempice

Visite in profondità su Albero non binario

Visite per la rappresentazione primo-figlio, fratello-successivo

```
// Solo ricorsiva
function PREORDER(TREE T)
    if T ≠ NIL then
        VISIT(T)
        PREORDER(T.first)
        PREORDER(T.next)
```

```
// Solo ricorsiva
function POSTORDER(TREE T)
    if T ≠ NIL then
        POSTORDER(T.first)
        POSTORDER(T.next)
        VISIT(T)
```

```
// Mista ricorsiva/iterativa
function PREORDER(TREE T)
    if T ≠ NIL then
        VISIT(T)
        tmp = T.first
        while tmp ≠ NIL do
            PREORDER(tmp)
            tmp = tmp.next
```

```
// Mista ricorsiva/iterativa
function POSTORDER(TREE T)
    if T ≠ NIL then
        tmp = T.first
        while tmp ≠ NIL do
            POSTORDER(tmp)
            tmp = tmp.next
        VISIT(T)
```

Costo (ottimo, pessimo, medio) in tutti i casi: $\Theta(n)$ (n = numero di nodi)

Visita in ampiezza su Albero non binario

Visita BFS per la rappresentazione primo-figlio, fratello-successivo

```
function BFS(Tree T)
    Let Q be a new Queue
    if T ≠ NIL then
        enqueue(Q, T)
    while Q.size ≠ 0 do
        x = dequeue(Q)
        visit(x)
        tmp = x.first
        while tmp ≠ NIL do
            enqueue(Q, tmp)
```

- Costo (ottimo, pessimo, medio): $\Theta(n)$ (n = numero di nodi)

Java (asdlab.libreria.Alberi.Albero)

Interfaccia Albero

```
public interface Albero {
  public int numNodi();

  public int grado(Nodo v);

  public Object info(Nodo v);

  public Nodo radice();

  public Nodo padre(Nodo v);

  public List figli(Nodo v);

  public List visitaDFS();

  public List visitaBFS();
  ...
}
```

Java (asdlab.libreria.Alberi.Nodo)

Classe astratta nodo

```
public abstract class Nodo implements Rif {

    // Il contenuto informativo associato a ciascun nodo
  public Object info;

  // Costruttore per l'istanziazione di nuovi nodi.
  public Nodo(Object info) {this.info = info;}

  // Restituisce il riferimento alla struttura dati
  // contenente il nodo.
  public abstract Object contenitore();
}

// asdlab.libreria.StruttureElem.Rif
public interface Rif {

}
```

Java (asdlab.libreria.Alberi.NodoBinPF)

Classe nodo per albero binario

```
public class NodoBinPF extends Nodo {
  public NodoBinPF padre; // Padre del nodo corrente.
  public NodoBinPF sin;   // Figlio sinistro del nodo corrente.
  public NodoBinPF des;   // Figlio destro del nodo corrente.
  public AlberoBin albero;// Albero cui il nodo appartiene.

  // Costruttore per l'istanziazione di nuovi nodi.
  public NodoBinPF(Object info) {super(info);}

  // Restituisce il riferimento alla struttura dati
  // contenente il nodo.
  public AlberoBin contenitore(){
    NodoBinPF n = this;
    while (n.padre != null) n = n.padre;
    return n.albero;
  }
}
```

Java (asdlab.libreria.Alberi.NodoPFFS)

Classe nodo per albero generico Primo figlio-Fratello Successivo

```
public class NodoBinPF extends Nodo {
  public NodoPFFS padre;// Padre del nodo corrente.
  public NodoPFFS primo;// Primo figlio del nodo corrente.
  public NodoPFFS succ; // Fratello successivo del nodo corrente.
  public Albero albero; // Albero cui il nodo appartiene.

  // Costruttore per l'istanziazione di nuovi nodi.
  public NodoPFFS(Object info) {super(info);}

  // Restituisce il riferimento alla struttura dati
  // contenente il nodo.
  public AlberoBin contenitore(){
    NodoPFFS n = this;
    while (n.padre != null) n = n.padre;
    return n.albero;
  }
}
```